Speculative Precomputation: Exploring the Use of Multithreading for Latency

Hong Wang, Microprocessor Research, Intel Labs
Perry H. Wang, Microprocessor Research, Intel Labs
Ross Dave Weldon, Logic Technology Development Group, Intel Corporation
Scott M. Ettinger, Microprocessor Research, Intel Labs
Hideki Saito, Software Solution Group, Intel Corporation
Milind Girkar, Software Solution Group, Intel Corporation
Steve Shih-wei Liao, Microprocessor Research, Intel Labs
John P. Shen, Microprocessor Research, Intel Labs

Index words: cache misses, memory prefetch, precomputation, multithreading, microarchitecture

ABSTRACT

Speculative Precomputation (SP) is a technique to improve the latency of single-threaded applications by utilizing idle multithreading hardware resources to perform aggressive long-range data prefetches. Instead of trying to explicitly parallelize a single-threaded application, SP does the following:

- Targets only a small set of static load instructions, called *delinquent loads*, which incur the most performance degrading cache miss penalties.
- Identifies the dependent instruction slice leading to each delinquent load.
- Dynamically spawns the slice on a spare hardware thread to speculatively precompute the load address and perform data prefetch.

Consequently, a significant amount of cache misses can be overlapped with useful work, thus hiding the memory latency from the critical path in the original program.

Fundamentally, contrary to conventional wisdom that multithreading microarchitecture techniques can be used to only improve the throughput of multitasking workloads or the performance of multithreaded programs, SP demonstrates the potential to leverage multithreading hardware resources to exploit a form of implicit thread-level parallelism and significantly speed up single-threaded applications. Most desktop applications in the traditional PC environment are not otherwise easily parallelized to take advantage of multithreading resources.

This paper chronicles the milestones and key lessons from Intel's research on SP, including an initial simulation-based evaluation of SP for both in-order and out-of-order multithreaded microarchitectures. We also look at recent experiments in applying software-based SP (SSP) to significantly speed up a set of pointer-intensive applications on a pre-production version of Intel® Xeon™ processors with Hyper-Threading Technology.

INTRODUCTION

Memory latency has become the critical bottleneck to achieving high performance on modern processors. Many large applications today are memory intensive, because their memory access patterns are difficult to predict and their working sets are becoming quite large. Despite continued advances in cache design and new developments in prefetching techniques, the memory bottleneck problem still persists. This problem worsens when executing *pointer-intensive* applications, which tend to defy conventional stride-based prefetching techniques.

®Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

™Xeon is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

One solution is to overlap memory stalls in one program with the execution of useful instructions from another program, thus effectively improving system performance in terms of overall *throughput*. Improving throughput of multitasking workloads on a single processor has been the primary motivation behind the emerging simultaneous multithreading (SMT) techniques [1][2][3]. An SMT processor can issue instructions from multiple hardware contexts, or *logical processors* (sometimes also called *hardware threads*), to the functional units of a super-scalar processor in the same cycle. SMT achieves higher overall throughput by increasing overall instruction-level parallelism available to the architecture via the exploitation of the natural parallelism between independent threads during each cycle.

However, this traditional use of SMT does not directly improve performance in terms of *latency* when only a single thread is executing. Since the majority of desktop applications in the traditional PC environment are single-threaded code, it is important to investigate if and how SMT techniques can be used to enhance single-threaded code performance by reducing latency.

At Intel Labs, extensive microarchitecture research efforts have been dedicated to discover and evaluate innovative hardware and software techniques to leverage multithreaded hardware resources to speed up single-threaded applications. One of the techniques is called Speculative Precomputation (SP), a novel thread-based cache prefetching mechanism. The key idea behind SP is to utilize otherwise idle hardware thread contexts to execute speculative threads on behalf of the main (non-speculative) thread. These speculative threads attempt to trigger future cache-miss events far enough in advance of access by the non-speculative thread that the memory miss latency can be masked. SP can be thought of as a special prefetch mechanism that effectively targets load instructions that exhibit unpredictable irregular or data-dependent access patterns. Traditionally, these loads have been difficult to handle via either hardware prefetchers [5][6][7] or software prefetchers [8].

In this paper, we chronicle several milestones we have reached including initial simulation-based evaluations of SP for both in-order and out-of-order multithreaded research processors [9][10][11][12][13][14], and highlight recent experiments in successfully applying software-based SP (SSP) to significantly speed up a set of pointer-intensive benchmarks on a pre-production version of Intel® Xeon™ processors with the Hyper-Threading Technology.

We first recount the motivation for SP, and we introduce the basic algorithmic ingredients and key optimizations, such as chaining triggers, which ensure the effectiveness of SP. We then compare SP with out-of-order execution, the traditional latency tolerance technique, and shed light on the effectiveness of combining both techniques. We follow with a discussion of the trade-offs for hardware-based SP and software-based SP (SSP), and in particular, highlight an automated post-pass binary adaptation tool for SSP. This tool can achieve performance gains comparable to that of implementing SSP using hand optimization. We then describe recent experiments where SSP is applied to speed up a set of applications on a pre-production version of Intel Xeon processors with the Hyper-Threading Technology. Finally, we review related work.

SPECULATIVE PRECOMPUTATION: KEY IDEAS

Chronologically, the key ideas for Speculative Precomputation (SP) were developed prior to the arrival of silicon for the Intel® Xeon™ processors with Hyper-Threading Technology. Our initial research work on SP was conducted on a simulation infrastructure modeling a range of research Itanium™ processors that support Simultaneous Multithreading (SMT) with a pipeline configurable to be either in-order or out-of-order. Before we discuss the trade-offs for hardware- vs. software-based implementations of SP, our discussion will assume the research processor model described below in

Table 1. We use a set of benchmarks selected from SPEC2000 and the Olden suite, including *art*, *equake*, *gzip*, *mcf*, *health* and *mst*.

Table 1: Details of the research Itanium processor models

Pipeline Structure	In-order: 8-12-stage pipeline. Out-of-order: 12-16-stage pipeline.
Fetch	2 bundles from 1 thread, or 1 bundle from each of 2 threads.
Branch pred	2K-entry GSHARE. 256 entry 4-way

®Intel is registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

™Xeon and Itanium are trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Expansion	Private, per-thread, in-order 8 bundle expansion queue
Register Files	Private, per-thread register files. 128 integer registers, 128 FP registers, 64 predicate registers, 128 application registers
Execute Bandwidth	In-order: 6 instructions from one thread or 3 instructions from each of 2 threads Out-of-order: 18-instruction schedule window
Cache Structure	L1 (separate I and D): 16K 4-way, 8-way banked, 1-2-cycle L2 (shared): 256K 4-way, 8-way banked, 7-14-cycle L3 (shared): 3072K 12-way, 1-way banked, 15-30-cycle Fill buffer (MSHR): 16 entries. All caches: 64-byte lines
Memory	115-230 cycle latency, TLB Miss Penalty 30 cycles.

Delinquent Loads

For most programs, only a small number of static loads are responsible for the vast majority of cache misses [15]. Figure 1 shows the cumulative contributions to L1 data cache misses by the top 50 static loads for the processor models in

Table 1 running benchmarks to completion. It is evident that a few poorly behaved static loads dominate cache misses in these programs. We call these loads *delinquent loads*.

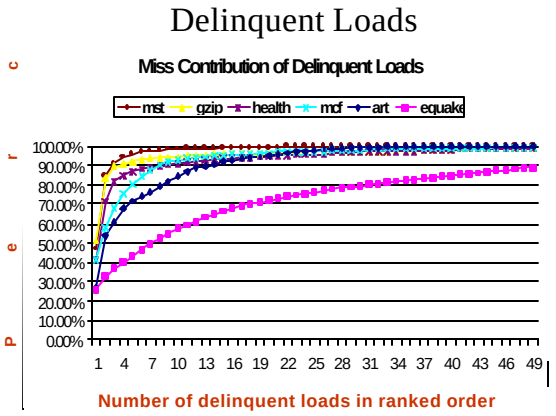

Figure 1: Cumulative L1 data cache misses due to delinquent loads

In order to gauge the impact of these loads on performance, Figure 2 compares the performance of a perfect memory subsystem, where all loads hit in the L1, to that of a memory subsystem that assumes the worst 10 delinquent loads always hitting in the L1 cache. In most cases, eliminating performance losses from only the top delinquent loads yields most of the speed-up achievable by the ideal memory. These data suggest that significant improvements can be achieved by just focusing latency-reduction techniques on the delinquent loads.

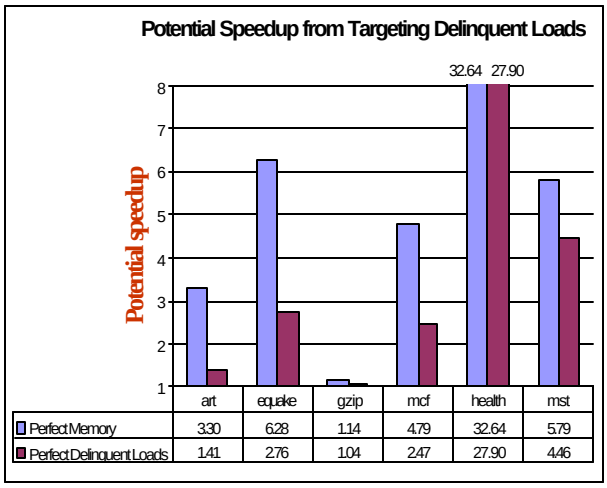

Figure 2: Speed-up when 10 delinquent loads are assumed to always hit in cache

SP Overview

To perform effective prefetch for delinquent loads, SP requires the construction of the *precomputation slices*, or p-slices, which consist of dependent instructions that compute the addresses accessed by delinquent loads. When an event triggers the invocation of a p-slice, a speculative thread is spawned to execute the p-slice. The speculatively executed p-slice then prefetches for the delinquent load that will be executed later by the main thread. Speculative threads can be spawned under one of two conditions: when encountering a *basic trigger*, which occurs when a designated instruction in the non-speculative thread is retired, or when encountering a *chaining trigger*, which occurs when a speculative thread explicitly spawns another.

Spawning a speculative thread entails allocating a hardware thread context, copying necessary live-in values into its register file, and providing the thread context with the address of the first instruction of the p-slice. If a free hardware context is not available, the spawn request is ignored.

Necessary live-in values are always copied into the thread context when a speculative thread is spawned. This eliminates the possibility of inter-thread hazards, where a

register is overwritten in one thread before a child thread has read it. Fortunately, as shown in Table 2, the number of live-in values that must be copied is very small.

Table 2: Slice statistics

Benchmark	Slices (#)	Average size (#inst)	Average # live-in
art	2	4	3.5
equake	8	12.5	4.5
gzip	9	9.5	6.0
mcf	6	5.8	2.5
health	8	9.1	5.3
mst	8	26	4.7

When spawned, a speculative thread occupies a hardware thread context until the speculative thread completes execution of all instructions in the p-slice. Speculative threads are not allowed to update the architectural state. In particular, stores in a p-slice are not allowed to update any memory state. For the benchmarks studied in this research, however, none of the p-slices include any store instructions.

SP Tasks

Several steps are necessary to employ SP: identification of the set of delinquent loads, construction of p-slices for these loads, and the establishment of triggers. In addition, upon dynamic execution with SP, proper control is necessary to ensure that the precomputation can generate timely and accurate prefetches. These steps can be performed by a variety of approaches including compiler assistance, hardware support, and a hybrid of both software and hardware approaches. These steps can be applied to any processor supporting SMT, regardless of differences in instruction set architectures (ISA) or pipeline organization. Different manifestations of SP are further discussed later in the paper.

Identify Delinquent Loads

The set of delinquent loads that contribute the majority of cache misses is determined through memory access profiling, performed either by the compiler or a memory access simulator [15], or by dedicated profiling tools for real silicon, such as the VTune™ Performance Analyzer [16]. From such profile analysis, the loads that have the largest impact on performance (i.e., incurring long latencies) are selected as delinquent loads. The total number of L1 cache misses can be used as the criterion to select delinquent loads, while other filters (e.g., L2 or L3 misses or total memory latency) could also be used. For example, in our simulation-based study, we use the L1 cache misses to identify the delinquent loads, while for our experiment on a pre-production version of the Intel Xeon processor with the Hyper-Threading Technology, we use L2 cache miss profiling from the VTune analyzer instead.

Construct and Optimize P-Slices

In this phase, a p-slice is created for each delinquent load. Depending upon the environment, the p-slice can be constructed by hand, via a simulator [11][13], by a compiler [14], or directly by hardware [12]. For example, a p-slice with a basic trigger can be captured via traditional backward slicing [17] within a window of dynamic instruction traces. By eliminating instructions that delinquent loads do not depend on, the resulting p-slices are typically of very small sizes, typically 5 to 15 instructions per p-slice. For p-slices with chaining triggers, a more elaborate construction process is required.

P-slices containing chaining triggers typically have three parts–a prologue, a spawn instruction for spawning another copy of the p-slice, and an epilogue. The prologue consists of instructions that compute values associated with a loop-carried dependency, i.e., those values produced in one loop iteration and used in the next loop iteration, such as updates to a loop induction variable. The epilogue consists of instructions that produce the address for the targeted delinquent load. The goal behind chaining trigger construction is for the prologue to be executed as quickly as possible, so that additional speculative threads can be spawned as quickly as possible.

To add chaining triggers to p-slices targeting delinquent loads within loops, the algorithm for capturing p-slices using basic triggers can be augmented to track the distance between different instances of a delinquent load. If two instances of the same p-slice are consistently spawned within a fixed-sized window of instructions, we create a new p-slice that includes a chaining trigger that targets the same delinquent load. Instructions from one

™ VTune is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

slice that modify values used in the next p-slice are added to the prologue. Instructions that are necessary to produce the address loaded by the delinquent load are added to the epilogue. Between the prologue and epilogue, a spawn instruction is inserted to spawn another copy of this same p-slice.

Condition Precomputation

To be effective, SP-based prefetches must be accurate and timely. By accuracy, we mean a p-slice upon spawning should use mostly valid live-in values to produce a correct prefetch address. By timeliness, we mean the speculative threads performing the SP prefetch thread should run neither behind nor too far ahead of the main non-speculative thread.

For accuracy, if spawning of the speculative thread is done only after its corresponding trigger reaches the commit stage of the processor pipeline, then the live-in values of the associated p-slice are usually guaranteed to be architecturally correct, thus ensuring precomputation will produce the correct prefetch address. An alternative policy might attempt to spawn as soon as the trigger instruction is detected at the decode stage of the pipeline. The drawback of such an early spawning scheme is that both the trigger and the live-in values are speculative and prefetching from the wrong address can occur.

For timeliness, basic trigger by definition is more sensitive to how far it is between the trigger and the target delinquent load and how long the p-slice is, since the thread spawning is tightly coupled to progress made by the main thread. Any overhead associated with thread spawning will not only reduce the headroom for prefetch but also incur additional latency on the main thread.

The use of chaining, while decoupling thread spawning from progress made by the main thread, could potentially be overly aggressive in getting too far ahead and evicting useful data from the cache before the main thread has accessed them. To condition the run-ahead distance between the main thread and the SP threads, a structure called an *Outstanding Slice Counter* (OSC), is introduced to track, for a subset of distinct delinquent loads, the number of speculative threads that have been spawned relative to the number of instances of a delinquent load that have not yet been retired by the non-speculative thread. Each entry in the OSC tracking structure contains a counter, the instruction pointer (IP) of a delinquent load and the address of the first instruction in a p-slice, which identifies the p-slice. This counter is decremented when the non-speculative thread retires the corresponding delinquent load, and is incremented when the corresponding p-slice is spawned. When a speculative thread is spawned for which the entry in the OSC is negative, the resulting speculative thread is forced to wait in the pending state until the counter becomes positive, during which time it is not considered for assignment to a hardware thread context.

As we will see later, the controlling mechanism can also be implemented entirely in software as part of the speculative thread.

SP Trade-offs

One of the key findings in our SP research is that the chaining trigger, assuming fairly conservative hardware support but with a proper conditioning mechanism, can be much more effective than the basic trigger even assuming ideal hardware support. The trade-offs between the basic trigger and the chaining trigger can be summarized as follows.

Basic Trigger With Ideal Hardware Assumption

Figure 3 shows the performance gains achieved through two rather ideal SP configurations. One is more aggressive in that speculative threads are spawned from the non-speculative thread at the rename stage, but only by an instruction on the correct control flow path using oracle knowledge. The other is a less aggressive one, in that speculative threads are spawned only at the commit stage, when the instruction is guaranteed to be on the correct path. In both cases, we assume aggressive and ideal hardware support for directly copying live-in values from the non-speculative parent thread's context to its child thread's context, i.e., one-cycle *flash-copy* of live-in values. This allows the speculative thread to begin execution of a p-slice just one cycle after it is spawned.

For each benchmark, results are grouped into three pairs, corresponding to, from left to right, 2, 4, and 8 total hardware thread contexts. Within each pair, the configuration on the left corresponds to spawning speculative threads in the rename stage, while the configuration on the right corresponds to spawning in the commit stage as described above.

Basic Trigger Without Ideal Hardware Assumption

We propose a more realistic implementation of SP, which performs thread spawning after the trigger instruction is retired and assumes overhead, such as potential pipeline flush and multiple-cycle transfer of live-in values across threads via memory. This approach differs from the idealized hardware approach in two ways. First, spawning a thread is no longer instantaneous. It will slow down the non-speculative thread, due to the need to invoke and execute the handler code to check hardware thread availability and copy out live-in values to memory to prepare for cross-thread transfer. At the very minimum, invoking this handler requires a pipeline flush. The

second difference is that p-slices must be modified with a prologue to first load their live-in values from the transfer memory buffer, thus delaying the beginning of precomputation.

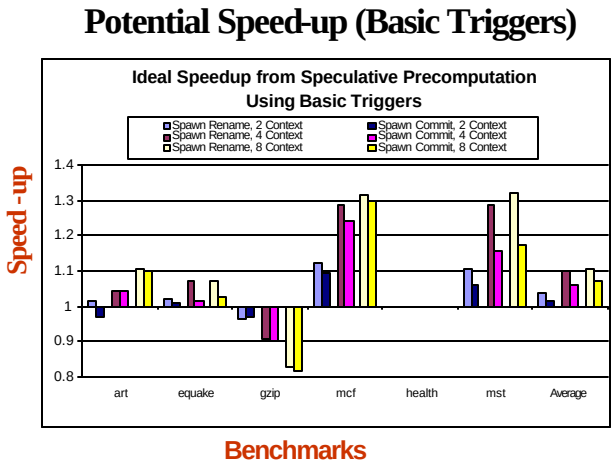

Figure 3: SP speed-up with basic trigger and ideal hardware assumptions

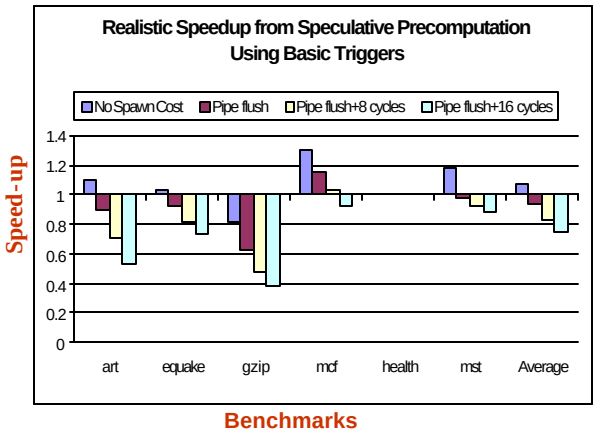

Figure 4: SP speed-up with basic trigger and realistic hardware

Figure 4 shows the performance speed-ups achieved when this more realistic hardware is assumed for a processor with eight hardware thread contexts. Four processor configurations are shown, each corresponding to differing thread-spawning costs. The leftmost configuration is given for reference, in which speculative threads are spawned with no penalty for the non-speculative thread, but must still perform a sequence of load instructions to read their live-in values from the memory transfer buffer. This configuration yields the highest possible performance because the main thread is still instantaneous in spawning a speculative thread. In the other three configurations, spawning a speculative thread causes the non-speculative thread's instructions following the trigger to be flushed from the pipeline. In the configuration second from the left, this pipeline flush is the only penalty, while in the third and fourth configurations, an additional penalty of 8 and 16 cycles, respectively, is assumed for the cost of executing the handler code to perform the live-in transfer.

Comparing these results to the performance of SP with ideal hardware (see Figure 3), the results for realistic SP in Figure 4 are rather disappointing. The primary reason that this performance falls short of that in the ideal case is the overhead incurred when the non-speculative thread spawns speculative threads. Specifically, the penalty of pipeline flush and the cost of performing live-in spill instructions in the handler both negatively affect the performance of the non-speculative thread.

Chaining Trigger

Figure 5 shows the speed-up achieved from realistic SP using chaining triggers as the number of thread contexts is varied. We assume that a thread spawning incurs a pipeline flush and an additional penalty of 16 cycles. Chaining triggers make effective use of available thread contexts when sufficient memory parallelism exists, resulting in impressive average performance gains of 51% with four threads and 76% with eight threads.

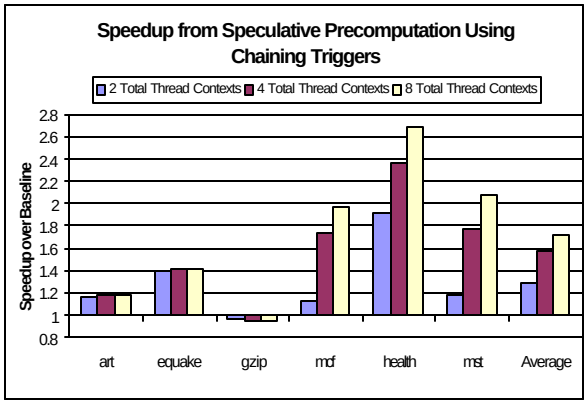

Figure 5: SP speed-up with chaining trigger and realistic hardware

Most noticeable is *health*. Though it does not benefit significantly from basic triggers (as shown in Figure 4) the speed-up is boosted to 169%, when using chaining triggers.

Figure 6 shows which level of the memory hierarchy is accessed by delinquent loads under three processor configurations: the baseline processor without use of SP, a processor with 8 thread contexts that uses basic triggers,

and a processor with 8 thread contexts that uses both basic and chaining triggers.

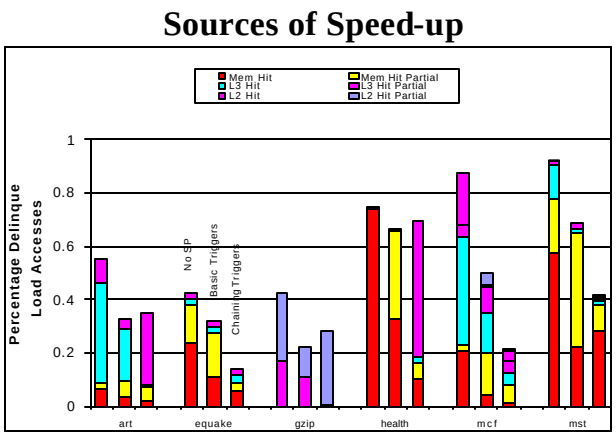

Figure 6: Reduction of cache misses in the memory hierarchy via SP-based prefetching

In general, basic triggers provide high accuracy but fail to significantly impact the number of loads that require access to the main memory. Even though basic triggers can be effective in targeting delinquent loads with relatively low latency, such as L1 misses, they are not likely to significantly help prefetch cache misses to main memory in a timely manner.

Chaining triggers, however, can achieve higher coverage and prefetch data in a much more timely manner, even for data that require access to the main memory. This is due to the chaining trigger's ability to effectively target delinquent loads and perform prefetches significantly far ahead of the non-speculative thread.

MEMORY LATENCY TOLERANCE: SP VS. OOO

Before the advent of thread-based prefetch techniques like SP, out-of-order (OOO) execution [18][19][20] has been the primary microarchitecture technique to tolerate cache miss latency. With the register renamer and reservation stations, an OOO processor is able to dynamically schedule the in-flight instructions, and execute those instructions independent of the missing loads, while the misses are being served.

Fundamentally, both OOO and SP aim to hide memory latency by overlapping instruction execution with the service to outstanding cache misses. OOO tries to overlap the outstanding cache-miss cycles by finding independent instructions after the missing load and executing them as early as possible, while SP prefetches for the delinquent loads far ahead of the non-speculative thread, thus overlapping future cache misses with the current execution of the non-speculative thread.

While both SP and OOO can reduce the data cache miss penalty incurred on the program's critical path, they differ in the targeted memory access instructions and the effectiveness for different levels of the cache hierarchy. On the one hand, while OOO can potentially hide the miss penalty for all load and store instructions to all layers of the cache hierarchy, it is most effective in tolerating L1 miss penalties. But for misses on L2 or L3, OOO may have difficulty in finding sufficient independent instructions to execute and overlap the much longer cache-miss latency. On the other hand, SP by design targets only a small set of delinquent loads that incur cache misses all the way to the memory.

To quantify the difference between SP and OOO, using the research processor models in Table 1, we evaluate two sets of benchmarks, one representing CPU-intensive workloads, including *gap*, *gzip* and *parser*, from SPEC2000Int, and the other representing memory-access-intensive workloads, including *equake* from SPEC2000fp, *mcf* from SPEC2000int, and *health* from the Olden suite.

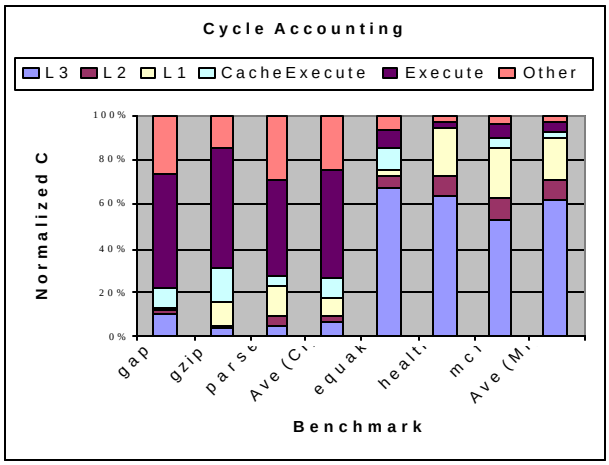

Figure 7: Characteristics of CPU-intensive vs. memory-intensive workloads on an in-order machine

Figure 7 depicts the cycle breakdown of these benchmarks on the in-order baseline processor. A cycle is assigned to *L1*, *L2*, and *L3* when the memory system is busy servicing the miss at the respective layer of cache hierarchy. *Execute* indicates that the processor issues an instruction for execution while the memory system is idle. Finally, *CacheExecute* shows the overlapping of cache misses with instruction execution. Clearly, the compute-intensive benchmarks spend most of their time in *Execute* while the memory-intensive benchmarks spend their time in waiting for cache misses.

Figure 8 shows speed-ups over the baseline model achieved by each of the two memory-tolerance techniques and by a combination of the two. The OOO processor model has four additional pipe stages to account for the increased complexity. Furthermore, SP assumes the use of chaining triggers and support for conditional precomputation.

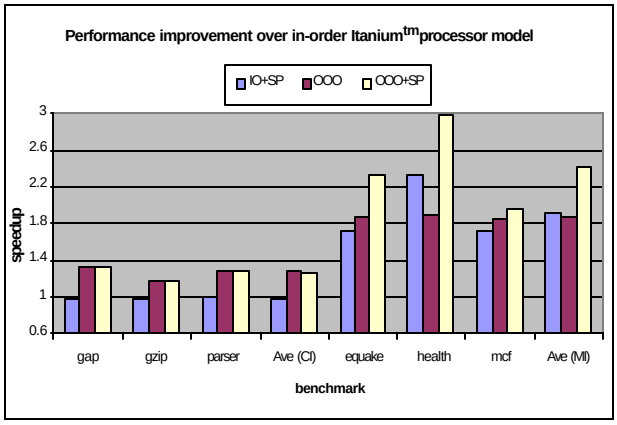

Figure 8: Speed-ups of in-order+SP, OOO, OOO+SP over in-order

Figure 9 further shows the cycle breakdown normalized to the in-order execution. This allows us to dissect where the speed-ups come from in terms of contributions leading to latency reduction.

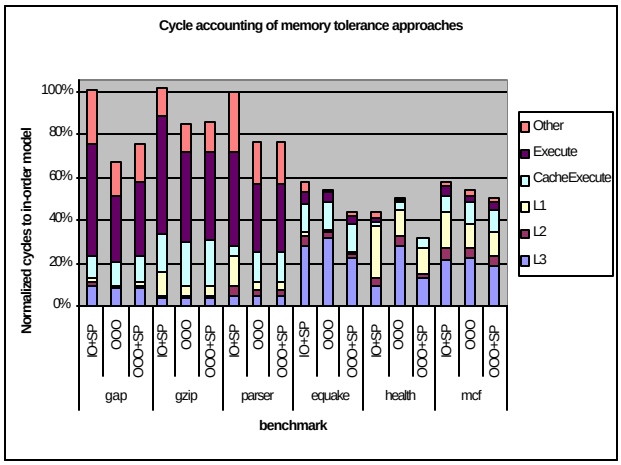

Figure 9: Cycle breakdown of in-order+SP, OOO and OOO+SP relative to in-order (100%)

The key findings can be summarized as follows.

OOO vs. SP

As shown in Figure 8, for memory-intensive workloads, the SP-enabled in-order SMT processor, albeit targeting only up to the top ten most delinquent loads that miss frequently in the L2 or L3 caches, can achieve slightly better speed-up than OOO. As shown in Figure 9, the speed-up is due to the reduction of the miss penalty at different levels of the cache hierarchy. For example, for *health*, OOO reduces the L3 cycle count from 62% in the baseline in-order to 28%, while SP achieves an even bigger reduction, down to 9%.

However, for compute-intensive benchmarks, SP can actually degrade performance. This is because for these benchmarks, almost all the delinquent loads that miss L1 hit in L2 and leave little headroom for the SP threads to run ahead and produce timely prefetches. In addition, spawning threads increase resource contention with the main thread and potentially can induce slowdowns in the main thread as well.

However, OOO is able to tolerate cache misses at all levels of the cache hierarchy and tolerate long latency executions on functional units. For instance, for *parser*, OOO can achieve a 10% reduction in the L1 cache stall cycles, and an even larger reduction of 12% in the execution cycles accounted by *Execute*. Furthermore, *CacheExecute*, the portion accounting for overlapping between cache servicing and execution, also increases by 9%.

Combination of OOO and SP

As shown in Figure 8, for compute-intensive benchmarks, SP does not bring about any speed-up beyond using OOO alone.

For memory-intensive benchmarks, however, the effectiveness of combining SP with OOO depends on the benchmarks. For *health*, if used individually, the OOO and SP approaches can achieve about a 131% and 90% speed-up, respectively. Together the two approaches achieve a near additive speed-up of 198%, demonstrating a potential complementary effect between the two approaches. Data in Figure 9 further shed light on the cause behind this effect. For *health*, SP alone can reduce *L3* cycles to 9% without improving *L1*, and OOO alone can reduce *L1* to 11% with a relatively smaller reduction in *L3*. By attacking both L1 and L3 cache misses, SP and OOO used in combination can achieve an overall reduction for both *L1* and *L3*. This is the root of the complementary effect between OOO and SP, where each covers cache misses at relatively disjointed levels of the cache hierarchy. Another interesting observation is that on the SP-enabled OOO processor, almost all instruction executions are overlapped with memory accesses, a desired effect of memory tolerance techniques.

For *mcf*, comparing the SP-enabled in-order execution (a.k.a. in-order+SP) with OOO in Figure 9, a relatively smaller difference exists between cycle counts in each

corresponding category. This is a clear indication of overlapping, whose root cause is the fact that SP and OOO redundantly cover the delinquent loads in the loop body.

A key to effectively utilizing SP on an OOO processor is to avoid overlapping the efforts of these two approaches. In particular, in typical memory-intensive loops, lengthy loop control that contains pointer chasing usually is on the critical path for the OOO processor. Loop control consists of instructions that resolve loop-carried dependencies and compute the induction variables for the next loop iteration. Once such a computation in the loop control is completed, independent instructions across multiple iterations can be effectively executed to tolerate cache misses incurred in the loop body of a particular iteration. A good combination of SP and OOO is to judiciously apply SP to perform prefetches for the critical loads in the loop control while letting OOO handle delinquent loads in the loop body. Then complementary benefits can be achieved, as shown in the case of *health*.

HARDWARE-ONLY SPECULATIVE PRECOMPUTATION VS. SOFTWARE-ONLY SPECULATIVE PRECOMPUTATION

The basic steps and algorithmic ingredients for Speculative Precomputation (SP) can be implemented in a gamut of techniques ranging from a hardware-only [12] approach to a software-only approach [14], in addition to the hybrid approaches originally studied in [11][13].

At one end of the spectrum, in close collaboration with Professor Dean Tullsen's research team at the University of California at San Diego, we investigated the hardware-only approach, called Dynamic Speculative Precomputation (DSP), a run-time technique that employs hardware mechanisms to identify a program's delinquent loads and generate precomputation slices to prefetch them. Like thread-based prefetching, the prefetch code is decoupled from the main program, allowing much more flexibility than traditional software prefetching. Like hardware prefetching, DSP works on legacy code and does not sacrifice software compatibility with future architectures and can operate on dynamic information rather than static to initiate prefetching and to evaluate the effectiveness of a prefetch. But unlike the software approaches, speculative threads on DSP are constructed, spawned, enhanced, and possibly removed by hardware. Both basic trigger- and chaining trigger-based p-slices can be efficiently constructed using a back-end structure off the critical path. Even with minimal p-slice optimization, a speed-up of 14% can be achieved on a set of various memory-limited benchmarks. More aggressive p-slice optimizations yield an average speed-up of 33%.

Interestingly, even in a multiprogramming environment where multiple non-speculative threads execute, if SP is applied to the worst behaving loads in the machine, regardless of which thread they belong to, the overall throughput can actually be improved, even if only one of the threads benefits directly from SP. In other words, though SP is originally intended to reduce the latency of a single-threaded application, it can also contribute to throughput improvement in a multiprogramming environment.

At the other end of the spectrum, we developed a post-pass compilation tool [14] that facilitates the automatic adaptation of existing single-threaded binaries for SSP on a multithreaded target processor without requiring any additional hardware mechanisms. This tool has been implemented in Intel's IPF production compiler infrastructure and is able to accomplish the following tasks:

1) Analyze an existing single-thread binary to generate prefetch threads.
2) Identify and embed triggering points in the original binary code.
3) Produce a new binary that has the prefetch threads attached, which can be spawned at run time.

The execution of the new binary spawns the prefetch threads, which are executed concurrently with the main thread. Initial results indicate that the prefetching performed by the speculative threads can achieve significant speed-ups on an in-order processor, ranging from 16% to 104%, on pointer-intensive benchmarks. Furthermore, the speed-ups achieved using the automated binary-adaptation tool loses at most 18% of the speed-up relative to that produced by hand-generated SSP code on the same processor.

To our knowledge, this is the first time that such an automated binary-adaptation tool has been implemented and shown to be effective in accomplishing the entire process of extracting dependent instructions leading to target operation, identifying proper spawning points, and managing inter-thread communication to ensure timely pre-execution leading to effective prefetches.

SPECULATIVE PRECOMPUTATION ON THE INTEL® XEON® PROCESSOR WITH HYPER-THREADING TECHNOLOGY

With the arrival of silicon for the Intel Xeon processor with Hyper-Threading Technology, it is of great interest to try out our Speculative Precomputation (SP) ideas on a real physical computer, since, thus far, our techniques have been primarily developed on simulation-based research processor models. Within just a few weeks of getting a system with a pre-production version of the Intel Xeon processor with Hyper-Threading Technology, we were able to come up with a crucial set of insights and innovative techniques to successfully apply software-only SP (SSP) to a small set of pointer-intensive benchmarks via hand adaptation of the original code. As shown in Table 3, significant performance boosts were achieved. The range of speed-ups per benchmark is due to the use of different inputs. This result was first disclosed in the 2001 Microprocessor Forum [2] where the details of Intel's Hyper-Threading Technology were originally introduced.

Benchmark	Description	Speed-up
Synthetic	Graph traversal in large random graph simulating large database retrieval	22% - 45%
MST (Olden)	Minimal Spanning Tree algorithm used for data clustering	23% - 40%
Health (Olden)	Hierarchical database modeling health care system	11% - 24%
MCF (SPEC2000int)	Integer programming algorithm used for bus scheduling	7.08 %

Table 3: Initial performance data: SP on a pre-production version of an Intel® Xeon™ processor with Hyper-Threading Technology

®Intel and Pentium are registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

™Xeon and VTune are trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

The silicon used in our experiment is the first generation implementation of Hyper-Threading Technology. The chip provides two hardware thread contexts and runs under Microsoft's Windows* XP Operating System optimized for Hyper-Threading Technology. The two hardware contexts are exposed to the user as two symmetric multiprocessing logical processors. The on-chip cache hierarchy has the same configuration as the commercially available Intel Pentium® 4 processor in the 2001 timeframe. The entire on-chip cache hierarchy is shared between two hardware threads. There is no special hardware support for SP on this chip. In the following subsections, we use a pseudo-code of the synthetic benchmark in Table 3 as an example to highlight the methodology of applying SSP.

Figure 10 shows the pseudo-code for this microbenchmark. Figure 11 and Figure 12 illustrate the pseudo-code for both the main thread and the SP prefetch worker thread.

```
1 main()
  {
22   n = NodeArray[0]
3   while(n and remaining)
    {
4      work()
5      n->i = n->next->j + n->next->k + n->next->l
6      n = n->next
7      remaining--
    }
  }
```

Line 4: 49.47% of total execution time
Line 5: 49.46% of total execution time
Line 5: 99.95% of total L2 misses

Figure 10: Pseudo-code for single-thread code and the delinquent load profile

Like the general SP tasks described earlier, our experiment consists of methodologies for identification of delinquent loads, construction of SP threads, embedding of SP triggers, and a mechanism enabling live-in state transfer between the main thread and the speculative thread.

The identification of delinquent loads can be performed with the help of Intel's VTune™ Performance Analyzer 6.0 [16]. For instance, as shown in Figure 10, the pointer de-referencing loads originated at Line 5 are identified as delinquent with regard to L2 misses, and they incur significant latency.

*Other brands may be claimed as the property of others.

Without explicit hardware support for thread spawning, for inter-thread communication and state transfer, we use standard Win32* thread APIs. **`CreateThread()`** is used to create the SP thread at initialization, **`SetEvent()`** is used to embed a basic trigger inside the main thread, and **`WaitForSingleObject()`** is used in the SP prefetch thread to implement the event-driven activation inside the corresponding speculative thread. In addition, we use global variables as a medium to explicitly implement inter-thread state transfer, where the main thread is responsible for copying out the live-in values before signaling a trigger event (using **`SetEvent()`**). The SP prefetch thread is responsible for copying in the live-in values prior to performing pointer-chasing prefetches.

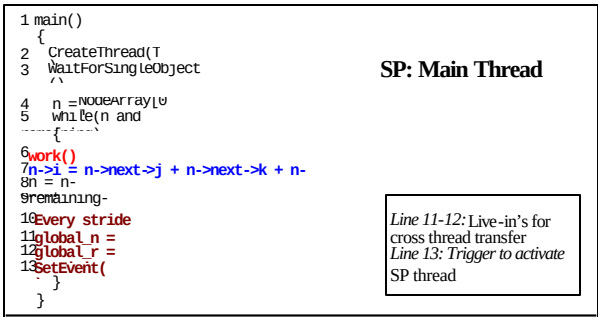

Figure 11: SP main thread pseudo-code

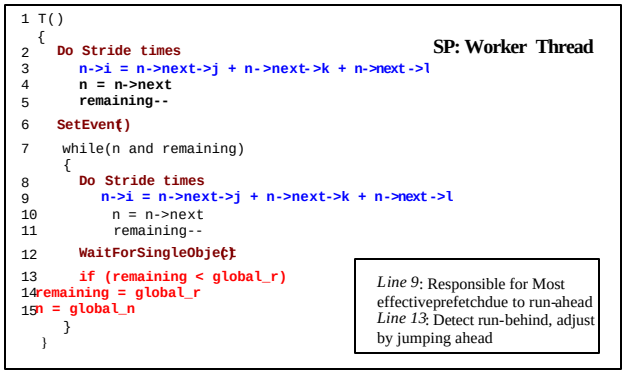

Figure 12: SP Prefetch worker thread–pseudo-code

Furthermore, as shown in Figure 12, a simple yet extremely important mechanism is used to implement SP conditioning inside the SP prefetch worker thread. This mechanism effectively ensures the SP prefetch worker thread performs the following two essential steps.

1. Upon each activation, it always runs a set of "stride" iterations of pointer chasing independent from the main thread.

   It is important to note that the pointer chasing loop bounded by "`stride`" effectively realizes a chaining trigger mechanism, since the progress can be made across multiple iterations independent of the main thread's progress.

2. After completing each set of "`stride`" iterations, it always monitors the progress made by the main thread to determine whether it is behind.

   If running behind, the SP thread will try to catch up with the main thread by synchronizing the global pointer.

   In addition, conditioning code can be introduced to detect if the SP thread is running too far ahead. The thread local variables "`remaining`" within both the main thread and the SP worker thread, are essentially trip counts recording their respective progress.

It is interesting to note that the SP worker thread uses only a regular load instruction and achieves effective prefetch for the main thread without actually using any literal prefetch instruction.

To do a fair comparison of performance, we use the Win32 API routine **`timeGetTime()`** to measure and compare the absolute wall clock execution time of the original code and the SSP-enabled code, both built for maximum speed optimizations using the Intel IA-32 C/C++ compiler [35]. For the example micro-benchmark, Figure 13 summarizes the reason why SSP-enabled code runs faster, using profiling information from the VTune Performance Analyzer 6.0 [16]. In short, the SP thread is able to prefetch successfully most cache misses for the identified delinquent loads. This optimization brings about a 22% – 45% speed-up for a range of input sizes.

Main Thread
- *Line 7* corresponds to *Line 5* of single thread code
  - Execution time:
    19% vs 49.46% in single-thread code
  - L2 miss:
    0.61% vs 99.95% in single-thread code

SP worker thread:
- *Line 9*:
  - Execution time:
    26.21%
  - L2 miss:
    97.61%

SP successful in shouldering most L2 cache misses

Figure 13: Why SSP-enabled code runs faster

* Other brands may be claimed as the property of others.

This successful experiment not only serves to corroborate insights and benefits of SP learned from our earlier studies, which were based on simulation, but also convincingly demonstrates an alternative way to effectively use multithreading processor resources, i.e., exploit a pseudo form of "thread-level parallelism" within the single-threaded application, and use multithreading hardware to reduce its latency.

RELATED WORK

Earlier ideas exploring speculative threads to achieve benefits of cache prefetches include Chappel et al., Simultaneous Subordinate Microthreading (SSMT) [27]; Sundaramoorthy et al., Slipstream Processors [28]; and Song et al., Assisted Execution [29].

Along with our research on SP [9][10][11][12][13][14], several thread-based prefetch paradigms have recently been proposed, including Zilles and Sohi's Speculative Slices [21], Roth and Sohi's Data Driven Multithreading [22], Luk's Software Controlled Pre-Execution [23], Annavaram et al., Data Graph Precomputation [24], and Moshovos et al., Slice-Processors [25]. Most of these techniques are equivalent to the basic trigger SP mechanism.

As pointed out by Roth et al. in [30], these thread-based prefetching approaches are in effect performing a logical form of access execute decoupling as originally envisioned by Smith in [31] and further studied in [32][33][34]. Instead of assuming a dedicated decoupled memory access engine, the access function is carried out by the prefetching SP threads. Using the post-pass SSP tool, special "access" threads are attached to the original code. "Access" and "Execute" threads are performed and overlapped ("pipelining") on distinct hardware thread contexts in a general-purpose SMT or CMP processor.

What distinguishes our research from other research in this area includes the discovery of the chaining trigger mechanism; in-depth analysis of trade-offs between different memory tolerance techniques, especially SP and OOO; a fully automated post-pass compilation tool for binary adaptation to enable SSP; and the physical experiment successfully demonstrating that using SSP on real hardware enabled with Hyper-Threading Technology can bring about significant speed-up for single-threaded benchmarks.

CONCLUSION

In this paper we examine key milestones from Intel's research on Speculative Precomputation (SP), a technique that allows a multithreaded processor to use spare hardware contexts to spawn speculative threads to prefetch data well in advance of the main thread. Fundamentally, our research demonstrates Simultaneous Multithreading (SMT) processor resources can be used effectively to reduce the latency and enhance the performance of single-threaded applications.

Instead of relying on the existence of a multitasking or multiprogramming workload environment in which many threads run simultaneously on SMT processors to achieve better throughput, SP is geared towards latency reduction by extracting assist threads out of the targeted single-threaded application itself. One insight about SP is that the potential performance gain is dictated by the reduction of cache miss latency (which is likely to get worse as clock frequency increases) and not by the increased instruction execution throughput in an SMT processor. Executing a small number of instructions of an SP thread can result in latency reduction far greater than the latency required to execute the SP thread. In traditional multithreading of an application, the potential speed-up is bounded by the number of instructions that can be executed in the additional thread context.

As explained in [2], the arrival of Intel's Hyper-Threading Technology on the Intel® Xeon™ processor marks the beginning of a new era: the transition from instruction-level parallelism (ILP) to thread-level parallelism (TLP). Multithreading techniques can help both power and complexity efficiency in future microarchitecture designs. It is of great interest to us to continue to look for alternate (and potentially better) use of multithreading resources. To summarize: Speculative Precomputation (SP) in effect leverages resources intended for thread-level parallelism (TLP) to achieve greater memory-level parallelism (MLP). This in turn significantly improves the effective instruction-level parallelism (ILP) of traditional single-threaded applications.

ACKNOWLEDGMENTS

The Speculative Precomputation research team has received tremendous support from Justin Rattner, Intel Fellow and Director of Microprocessor Research in Intel Labs (formerly MRL), and Richard Wirt, Intel Fellow and general manager, Software Solution Group (SSG). Individuals from various organizations who have provided critical support include Kevin J. Smith, David Sehr, Wilfred

®Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

™Xeon is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Pinfold, Jesse Fang, George K Chen, Gerolf Hoflehner, Dan Lavery, Wei Li, Xinmin Tian, Sanjiv Shah, Ernesto Su, Paul Grey, Ralph Kling, Jim Dundas, Wen-hann Wang, Yong-fong Lee, Ed Grochowski, Gadi Ziv, Shalom Goldenberg, Shai Satt, Ido Shamir, Per Hammarlund, Debbie Marr, John Pieper, Bo Huang, Young Wang, Rob Portman, Pete Andrews, Tony Martinez, Oren Gershon, Jonathan Beimel, Ady Tal, Yigal Zemach, and Leonid Baraz.

Professor Dean M. Tullsen, and his research team at UC San Diego, have been our closest collaborators throughout our research into Speculative Precomputation. In particular, Mr. Jamison D Collins, a Ph.D. candidate of Professor Tullsen's, has made significant contributions.

Finally, we thank the referees of this paper for their useful suggestions. We are grateful to Lin Chao, Judith Anthony and Marian Lacey for their extremely careful editing.

REFERENCES

[1] J. Emer, "Simultaneous Multithreading: Multiplying Alpha's Performance," *Microprocessor Forum*, Oct 1999.

[2] G. Hinton and J. Shen, "Intel's Multi-Threading Technology," *Microprocessor Forum*, October 2001.

[3] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," in *22nd ISCA*, June 1995.

[4] D. M. Tullsen and J. A. Brown, "Handling Long-Latency Loads in a Simultaneous Multithreading Processor," in *Micro-34*, Dec. 2001, pp. 318-327.

[5] T. Chen, "An Effective Programmable Prefetch Engine for On-chip Caches," In *Micro-28*, pp 237-242, Dec. 1995.

[6] N. Jouppi, "Improving Direct-mapped Cache Performance by the Addition of a Small Fully associative Cache and Prefetch Buffers," in *ISCA-17*, pp. 364-373, May 1990.

[7] D. Joseph and D. Grunwald, "Prefetching using Markov Predictors," in *ISCA-24*, pp. 252-263, June 1997.

[8] T. Mowry and A. Gupta, "Tolerating Latency through Software-controlled Prefetching in Shared-memory Multiprocessors," in *Journal of Parallel and Distributed Computing*, pp. 87-106, June 1991.

[9] H. Wang, et al., "A Conjugate Flow Processor," in *Docket No. 884.225US1*, Patent Pending, May 2000.

[10] H. Wang, et al., "Software-based Speculative Precomputation and Multithreading," in *Docket No. 042390.P10811*, Patent Pending, March 2001.

[11] J. Collins, H. Wang, D. Tullsen, C, Hughes, Y-F Lee, D. Lavery, J. Shen. Speculative Precomputation: Long-range Prefetching of Delinquent Loads," in *28th ISCA*, July 2001.

[12] J. Collins, D. Tullsen, H. Wang, J. Shen, "Dynamic Speculative Precomputation," in *Micro-34*, pp. 306-317, December 2001.

[13] P. Wang, H. Wang, J. Collins, E. Grochowski, R. Kling, J. Shen, "Memory latency-tolerance approaches for Itanium processors: out-of-order execution vs. speculative precomputation," in *Proceedings of the 8th IEEE HPCA*, Feb 2002.

[14] S. S. W. Liao, P. Wang, H. Wang, G. Hoflehner, D. Lavery, and J. P. Shen, "Post-pass Binary Adaptation for Software-based Speculative Precomputation," Accepted for publication at *PLDI'02*.

[15] S. G. Abraham and B. R. Rau, "Predicting Load Latencies using Cache Profiling," in *HP Lab Technical Report HPL-94-110*, Dec 1994.

[16] Intel Corp. VTune Performance Analyzer. http://developer.intel.com/software/products/VTune/index.htm

[17] C. Zilles and G. Sohi, "Understanding the backward slices of performance degrading instructions," in *27th ISCA*, pp. 172-181, June 2000.

[18] D. P. Bhandarkar, *Alpha Implementations and Architecture*, Digital Press, Newton, MA, 1996.

[19] J. Heinrich, *MIPS R10000 Microprocessor User's Manual*, MIPS Technologies Inc., Sept 1996.

[20] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker and P. Roussel, "The Microarchitecture of the Pentium 4 Processor," *Intel Technology Journal*. Q1, 2001.

[21] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices, in *28th ISCA*, July 2001.

[22] A. Roth and G. Sohi, "Speculative Data-Driven Multithreading," in *7th HPCA*, Jan 2001.

[23] C. K. Luk, "Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors," in *28th ISCA*, June 2001.

[24] M. Annavaram, J. Patel, and E. Davidson, "Data Prefetching by Dependence Graph Precomputation," in *ISCA-28*, pp 52-61, July 2001.

[25] A. Moshovos, D. Pnevmatikatos, A. Baniasadi, "Slice processors: an implementation of operation-based prediction, in *International Conference on Supercomputing*, June 2001.

[26] A. Roth, A. Moshovos, and G. Sohi, "Dependence-based prefetching for linked data structures," in *ASPLOS-98*, pp. 115-126, Oct. 1998.

[27] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt, "Simultaneous Subordinate Microthreading (SSMT)," in *26th International Symposium on Computer Architecture*, May 1999.

[28] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream Processors: Improving both Performance and Fault Tolerance," in *9th ASPLOS*, Nov. 2000.

[29] Y. Song and M. Dubois, *Assisted Execution. Technical Report #CENG 98-25, Department of EE-Systems*, University of Southern California, Oct. 1998.

[30] A. Roth, C. B. Zilles, G. S. Sohi, "Microarchitectural Miss/Execute Decoupling," in *MEDEA Workshop*, Oct. 2000.

[31] J. E. Smith. "Decoupled Access/Execute Computer Architecture," in *9th ISCA*, July 1982.

[32] M. K. Farrens, P. Ng, and P. Nico, "A Comparison of Superscalar and Decoupled Access/Execute Architectures," in *26th Micro*, Nov. 1993.

[33] G. P. Jones and N. P. Topham, "A Limitation Study into Access Decoupling," in *3rd Euro-Par Conference*, Aug. 1997.

[34] J. M. Parcerisa and A. Gonzalez, "The Synergy of Multithreading and Access/Execute Decoupling," in *5th HPCA*, Jan. 1999.

[35] A. Bik, M. Girkar, P. Grey and X. Tian, "Efficient Exploitation of Parallelism on Pentium III and Pentium 4 Processor-Based Systems," in *Intel Technology Journal*, Q1, 2001. http://www.intel.com/technology/itj/q12001/articles/art_6.htm

AUTHORS' BIOGRAPHIES

Hong Wang is a senior staff engineer in Microprocessor Research in Intel Labs. His current interests are in discovering unorthodox ideas and applying them to future Intel processor designs. Hong joined Intel in 1995 and earned a Ph.D. degree in electrical engineering from the University of Rhode Island in 1996. His e-mail is hong.wang@intel.com.

Perry H. Wang joined Intel in 1995. He is with the Microprocessor Research Group in Intel Labs. His technical interests are in advanced microarchitectures and compiler optimizations. He received a B.S. degree in engineering physics and an M.S. degree in computer science from the University of Michigan in Ann Arbor. His e-mail is perry.wang@intel.com

R. David Weldon joined Intel in 2000. He currently works in the Logic Technology Development group, designing future IA-32 processors. David has a BSEE degree from the University of Washington and a Masters degree from Cornell University. His technical interests include processor microarchitecture and hardware/software co-design. His e-mail is ross.d.weldon@intel.com

Scott M. Ettinger joined Intel in 2001. His technical interests are in computer architecture and multimedia signal processing. He received a B.S. and an M.S. degree in electrical engineering from the University of Florida. His e-mail is scott.m.ettinger@intel.com

Hideki Saito received a B.E. degree in Information Science in 1993 from Kyoto University, Japan, and a M.S. degree in Computer Science in 1998 from University of Illinois at Urbana-Champaign, where he is currently a Ph.D. candidate. He joined Intel Corporation in June 2000 and has been working on multithreading and performance analysis. He is a member of the OpenMP Parallelization group. His e-mail is hideki.saito@intel.com

Milind Girkar received a B.Tech. degree from the Indian Institute of Technology, Mumbai, an M.S. degree from Vanderbilt University, and a Ph.D. degree from the University of Illinois at Urbana-Champaign, all in computer science. Currently, he manages the IA-32 compiler development team in Intel's Software Solution Group. Before joining Intel, he worked on a compiler for the UltraSPARC platform at Sun Microsystems. His e-mail is milind.girkar@intel.com.

Steve Shih-wei Liao received a B.S. degree in computer science from National Taiwan University, and M.S. and Ph.D. degrees in electrical engineering from Stanford University. His research interests are in program analyses and optimizations, computer architectures, and programming environments. He currently works in the Microprocessor Research Group at Intel Labs. His e-mail is shih-wei.liao@intel.com.

John P. Shen currently directs Microarchitecture Research in Intel Labs. Prior to joining Intel in 2000, he was on the faculty of the Electrical and Computer Engineering Department of Carnegie Mellon University for over 18 years. He is an IEEE Fellow and is currently writing a textbook on "Fundamentals of Superscalar Processor Design" which will be published by McGraw-Hill in 2002. His e-mail is john.shen@intel.com.

Copyright © Intel Corporation 2002. This publication was downloaded from http://developer.intel.com/ .

Legal notices at http://developer.intel.com/sites/developer/tradmarx.htm.